Simulation de mouvements de foule dans une zone aéroportuaire (phase 2)

Margaux Azéma, Maxime André, Cédric Zimmermann, Estelle Boudassou

Introduction 4

Contexte de l'étude 6

Support de l'étude 6

Cahier des charges 6

Choix du périmètre de travail 7

Traduction du code ADA en code JAVA 8

Traitement d'image 8

Figure 1 : Détection de portes par l'algorithme de Bresenham ***11***

Figure 2 : Gestion des segments de détection de sorties ***11***

Figure 3 : Algorithme de Bresenham ***13***

Figure 4 : Tracé de sortie primaire ***13***

Figure 5 : Position de la personne avec un rayon 1 ***14***

Figure 6 : Déplacement d'une personne avec un rayon 1 ***14***

Figure 7 : Déplacement d'une personne avec un rayon 2 ***15***

Figure 8 : Problème de déplacement avec un rayon 2 au niveau des sorties ***15***

Figure 9 : Tracé final adapté ***16***

Figure 10 : Extérieur de la salle aéroportuaire à transformer en noir ***16***

Figure 11 : Image traitée par l'algorithme de FloodFill ***17***

Conception de l'IHM et mécanismes rattachés 19

Figure 12 : Interface graphique ***19***

Figure 13 : Machine à états ***20***

Algorithme de Fast Marching et descente de gradient 22

Sauvegarde et récupération des données de simulation 23

Implémentation 24

Mise en oeuvre de l'algorithme de comportement Uzawa 24

Figure 14 : Représentation de la distance entre deux personnes i et j ***24***

Mise en oeuvre de l'algorithme d'itération Euler 25

Synthèse 25

Figure 15 : Synthèse de l'utilisation des algorithmes 25

Organisation et méthodes 26

Division du travail 26

Suivi du projet 27

Contraintes 27

Retour d'expérience 28

Mise en oeuvre de l'architecture 28

Tests 28

Matrice de conformité 30

Conclusion 33

Annexes 34

I. Annexe 1 : Aspects mathématiques de l'algorithme d'Uzawa 34

II. Annexe 2 : Récapitulatif des cas d'utilisation, des besoins et de leur niveau de priorité 37

Bibliographie 41

Introduction

État de l'art

La modélisation des mouvements de foule est au cœur de nombreux travaux, et connaît aujourd'hui un grand intérêt. En effet, la construction de bâtiments comme les salles aéroportuaires requiert certaines études de sécurité, basées sur des modèles théoriques destinés à évaluer les risques en cas d'évacuation d'urgence.
Les modélisations de telles situations permettent d'estimer des temps de sortie, mais aussi les lieux d'une zone les plus embouteillés. Plusieurs modèles permettent d'étudier de façon théorique les mouvements de foule, et la plupart utilisent une représentation microscopique de la foule : chaque personne est modélisée indépendamment, et on observe le résultat de l'évolution de chaque individu. Pour visualiser cette évolution, on peut modéliser la zone aéroportuaire sous la forme d'une grille, dont chacune des cases est soit vide, soit occupée par une personne.
D'une part chaque personne a une vitesse souhaitée, celle qu'elle aurait en l'absence des autres, d'autre part, la vitesse réelle des individus prend en compte une certaine contrainte d'encombrement maximal. Plus précisément , dans le modèle étudié, la vitesse réelle est la projection de la vitesse souhaitée sur un ensemble dit admissible (qui respecte une contrainte de non-chevauchement des disques représentant les individus). Le modèle, en choisissant une vitesse souhaitée particulière (celle dirigée par le plus court chemin évitant les obstacles) utilise la méthode reposant sur l'algorithme de Fast Marching, alors que celui choisissant la vitesse réelle comme projection de la vitesse souhaitée repose sur l'algorithme d'Uzawa.

Problématique

- Analyse de l'existant

Le but de ce projet est de réaliser des simulations de mouvements de foule en zone aéroportuaire. Le simulateur de mouvement de foule envisagé a déjà fait l'objet d'une première phase d'étude et de développement en langage ADA et permet notamment :

  - une définition des paramètres de simulation,
  - l'extraction d'une cartographie du lieu de simulation à partir d'une image,
  - la mise en application d'un algorithme de Fast Marching qui calcule les distances à la sortie afin de créer le champ de vitesse qui est suivi par la foule lors des simulations,
  - une méthode de descente de gradient qui calcule la direction des vecteurs vitesse en chaque point de la zone,
  - un algorithme de comportement basique utilisé pour vérifier le bon comportement du simulateur.

Pour plus de détails, veuillez-vous référer au rapport de projet ADA/IA *Simulation de mouvements de foules*.

Cette première phase de développement du simulateur ne demandait pas d'implémenter un algorithme de mouvement de foule réaliste, c'est pourquoi, seul un algorithme de validation a été mis en œuvre jusqu'ici.
De plus, en ADA, les dialogues avec l'utilisateur, pour la saisie de paramètres par exemple, étaient faits dans le terminal, ce qui n'était pas très intéractif.

- Travail à réaliser

L'objet du travail proposé pour la seconde phase du projet est dans un premier temps de traduire les algorithmes précédemment implémentés du code ADA en code JAVA, pour se concentrer par la suite sur la mise en oeuvre d'un algorithme de comportement particulier: l'algorithme d'Uzawa. Afin de rendre notre simulateur plus attrayant, les interactions entre le système et l'utilisateur seront améliorées à l'aide d'une interface graphique.

Ce travail à réaliser a été défini avec l'aide de nos tuteurs et à partir du nouveau cahier des charges (cf I.B. Cahier des charges). De plus, à l'aide d'une analyse effectuée sur le système existant en ADA, nous avons pu définir les différents cas d'utilisation ainsi que les besoins qui en découlent (**cf Annexe 2: Récapitulatif des cas d'utilisation, des besoins et de leur niveau de priorité**)

I. Contexte de l'étude

Le but de ce projet est de développer un outil de simulation de mouvements de foule basé sur le simulateur déjà implémenté en ADA, afin d'étudier par la suite le comportement des usagers dans une zone aéroportuaire.

A. Support de l'étude

Il ne s'agit pas de réétudier le problème du mouvement d'une foule dans son intégralité. Il s'agit d'exploiter et d'utiliser les algorithmes déjà mis en œuvre pour modéliser une foule et décrits dans notre précédent rapport de projet ADA/IA *Simulation de mouvements de foules*, afin d'accélérer l'étude du comportement des personnes dans la zone aéroportuaire.
Parmi les algorithmes de comportement possibles, c'est celui d'Uzawa qui a été choisi par nos clients/tuteurs et qui sera implémenté dans ce projet.
Ces notions de comportements et l'utilisation de l'algorithme d'Uzawa sont extrêmement bien détaillés dans le chapitre 5 de la thèse de doctorat d'université en mathématiques de Juliette Venel, intitulée *Modélisation mathématique et numérique de mouvements de foule*. Ce document sera considéré comme document support de notre projet.

B. Cahier des charges

Pour réaliser ce simulateur, plusieurs points principaux sont attendus, c'est à dire:

- **la réécriture en JAVA du logiciel développé en ADA** tout en lui conservant l'ensemble de ses fonctionnalités :
    - traitement d'image et extraction d'une cartographie de la salle aéroportuaire,
    - détection de la sortie,
    - placement aléatoire des personnes,
    - calcul des vitesses souhaitées par l'algorithme de Fast Marching,
    - performances,
    - sauvegarde de simulation.
- **la mise en oeuvre de l'algorithme de comportement** ayant servi à la validation du logiciel en phase 1 afin de s'assurer du fonctionnement à l'identique et éventuellement amélioré du logiciel après traduction en JAVA,
- **l'implémentation d'un algorithme d'évolution d'une foule** tel que décrit dans la thèse de Juliette Venel: l'algorithme d'Uzawa.

Travail complémentaire

Si le délai le permet, un traitement multi-agent sera effectué sur le premier algorithme de comportement utilisé pour la validation. Ce traitement ne doit pas être forcément éprouvé, il sera ultérieurement exploité avec des algorithmes plus élaborés (comme Uzawa) dans d'autres projets. Ainsi, l'algorithme de comportement doit être mis en œuvre de manière à ce qu'il puisse être remplacé facilement par un autre.

Contraintes supplémentaires

Le simulateur devra pouvoir sauvegarder dans des fichiers l'état d'avancement de chaque simulation à intervalle de temps régulier (intervalle défini de façon arbitraire). Ces fichiers seront datés et permettront par la suite, de redémarrer et visualiser une simulation interrompue sans avoir à la relancer depuis le départ ainsi que rejouer une simulation déjà visualisée afin d'éventuellement comparer les résultats de plusieurs simulations.

C. Choix du périmètre de travail

Pour ce qui est de la **définition des paramètres de simulation**, on prendra :

- une "*image d'entrée*" : après un traitement d'image adéquat (qui proviendra de l'extraction d'une cartographie du lieu de simulation) qui se fera avec l'aide de classe déjà implémentée en JAVA. On produira une matrice de coefficients qui représentera la zone aéroportuaire à étudier,
- le *temps de simulation* : la simulation pourra ainsi s'arrêter de manière automatique quand ce temps de simulation sera atteint,
- une *nombre initial de personnes* : celle-ci seront réparties dans un carré contenu dans la zone qui lui-même sera placé à l'initiative de l'utilisateur,
- le *nombre de sorties* : l'utilisateur choisira le nombre de sorties et pourra détecter leurs extrémités (au clic) sur les contours pour permettre le traitement de l'image par l'algorithme de Fast Marching.

Les images utilisées pourront être dessinées à la main puis scannées ou photographiées comme ça l'était dans le logiciel réalisé en ADA.

Choix du format des fichiers de sauvegarde utilisés

Le format de fichiers texte (d'extension .txt) sera utilisé pour stocker toutes les données nécessaires à la reprise de visualisation et à la sauvegarde d'une simulation interrompue.

II. Traduction du code ADA en code JAVA

Le traitement d'image sera fait et facilité grâce à des classes déjà prêtes en JAVA, l'algorithme de Fast Marching et l'algorithme de comportement basique seront réécrits intégralement. Les algorithmes traduits seront ensuite immédiatement testés.

A. Traitement d'image

1. Introduction

Pour pouvoir être pleinement exploitée par l'algorithme de Fast Marching et les algorithmes de comportements, l'image doit subir (tout comme en ADA) des traitements préalables.

L'idée d'une traduction « bête et méchante » des algorithmes écrits en ADA a vite été abandonnée au vu des possibilités qu'offre l'API Java 2D.

Il existe en JAVA une super classe *Image* fournie dans la JDK depuis sa version 1.0 : celle-ci est une classe abstraite qui permet de représenter les images sous forme d'un rectangle de pixels. Cependant cette dernière ne permet pas d'accéder aux pixels. Elle est donc inadaptée pour le traitement d'images (et donc inadaptée à notre problème).

C'est pourquoi nous avons utilisé la classe *BufferedImage* qui hérite de la classe *Image* et implémente ses interfaces mais permet en plus d'examiner l'intérieur des images chargées et travailler directement sur ses données. On peut donc à partir d'un objet *BufferedImage*, récupérer et changer les couleurs des pixels. Comme son nom l'indique, *BufferedImage* est une image tampon. Cette classe gère l'image dans la mémoire et fournit des méthodes pour l'interprétation des pixels. Si on veut faire du traitement d'images, il faut donc travailler avec des objets *BufferedImage*.

Partant de ces constatations, nous avons créé une classe ***NotreImage*** avec en attribut une *BufferedImage*. Toutes les opérations de traitement de l'image seront définies comme étant des méthodes de cette classe.

2. Les différents traitements d'image

a) La binarisation

Pour les besoins du Fast Marching, la salle où se déroule la simulation doit être décomposée en trois catégories de zones : les zones vierges (représentées par des pixels blancs), les obstacles (représenté par des pixels noirs), et les sorties (représentées par des pixels rouges) : ***cf. d) Détection de la (des) porte(s).***

La binarisation de l'image (dont le format de départ peut être un format JPG, PNG, GIF ou encore BMP) permet de n'avoir plus que deux couleurs au sein de l'image : du blanc et du noir. Le principe est de créer une *BufferedImage* de type *TYPE_BYTE_BINARY* au sein de laquelle on va venir recopier le contenu de l'image initiale au format RGB (*TYPE_INT_RGB*).

b) Le lissage

Pour éliminer toutes les aspérités de l'image (présentes si la photo de la salle de simulation a mal été prise), l'utilisateur doit, à sa convenance, lisser l'image, c'est-à-dire éliminer le bruit qui est apparu.

Pour réaliser ce lissage nous avons utilisé la classe *Kernel* (*java.awt.image.Kernel*) et la classe *ConvolveOp* (*java.awt.image.ConvolveOp*) pour réaliser une convolution de l'image, c'est-à-dire lui appliquer un « flou ».

c) Agrandissement (zoom plus) et réduction (zoom moins)

Selon la taille initiale de l'image, l'utilisateur peut avoir envie de réduire ou d'agrandir l'image pour arriver au niveau de zoom qu'il désire pour visualiser la simulation. Pour réaliser les opérations de zoom nous avons utilisé la classe *AffineTransformOp* (*java.awt.image.AffineTransformOp*) et effectué une **interpolation bicubique** de l'image initiale (java.awt.AffineTransformOp.TYPE_BICUBIC). En effet, lorsqu'on zoome, il est nécessaire de créer des pixels : l'interpolation consiste à ajouter des pixels là où il n'y en a pas, ou au contraire à en enlever en cas de dé-zoom.

Principe de l'interpolation :

Notre but était de perdre le moins de qualité possible lors du zoom et dé-zoom de l'image à traiter.
L'interpolation utilise les pixels connus de l'image, et estime des pixels intermédiaires pour les pixels manquants. Pour créer un nouveau pixel en cas de zoom, il suffit de trouver la meilleure estimation possible de la couleur et de l'intensité d'un pixel en se basant sur les valeurs des pixels qui l'entourent. Sans interpolation, le traitement d'image consiste uniquement à agrandir les pixels et on perd ainsi en qualité. Nous avons réfléchi à deux méthodes d'interpolation :

- *L'interpolation bilinéaire*, qui prend en compte les quatre pixels les plus proches du pixel inconnu et fait une moyenne des valeurs de ces quatre pixels pour interpoler le pixel manquant.

- *L'interpolation bicubique*, qui, quant à elle va plus loin que l'interpolation bilinéaire : les seize pixels les plus proches du pixel à interpoler sont pris en compte, tout en donnant une importance plus grande à ceux qui sont les plus proches du pixel à interpoler. Nous avons choisi cette méthode car c'est celle qui donne les meilleurs résultats en terme de qualité et en temps.

Ces deux principes sont expliqués sur le site VIRUSPHOTO, "Comprendre l'interpolation numérique".

d) Détection de la (des) porte(s)

Contrairement au projet ADA, en JAVA le choix a été fait de ne pas poser de restrictions quant au nombre et au positionnement des portes, nous avons donc dû revoir la méthode utilisée pour les détecter dans le cadre de la première phase du projet.

Ainsi nous avons décidé de laisser l'utilisateur cliquer directement sur l'image pour renseigner le point d'entrée et de sortie de chaque porte (via une interface graphique, les données récoltées sont ensuite stockées dans les attributs d'une classe particulière nommée ***Porte***).

Grâce aux coordonnées récupérées, nous avons pu « tracer un trait » entre les deux extrémités détectées à la main (c'est-à-dire changer les pixels de la sortie en rouge). Pour cela **l'algorithme de Bresenham** a été utilisé : il permet le tracé d'une droite entre deux points en un minimum de pixels dit « allumés » (on « allume » les pixels passant le plus près possible de la ligne idéale).

Fonctionnement de l'algorithme de Bresenham utilisé :

(x_i,y_i) et (x_f,y_f) représentent respectivement le point d'entrée et le point de sortie de la porte :

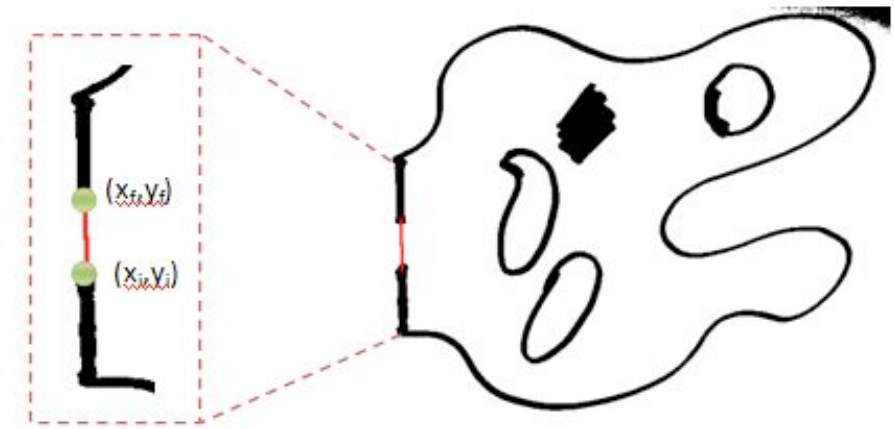

Figure 1 : Détection de portes par l'algorithme de Bresenham

On utilise deux variables pour gérer des incréments de 1 ou -1 pour x variant de x_i à x_f et y variant de y_i à y_f.

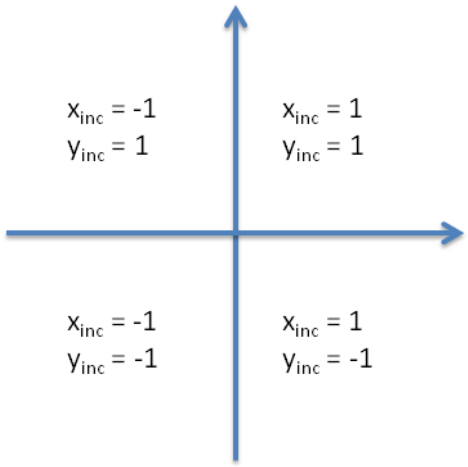

Figure 2 : Gestion des segments de détection de sorties

x_{inc} et y_{inc} donnent la tendance du segment (segment avec une pente positive, négative, vers les x croissants, vers les x décroissants …).

On a deux parties alternatives dans l'algorithme :

- une pour incrémenter en x si le segment a tendance à être "plutôt horizontal",

- l'autre pour l'incrémenter en y si le segment a tendance à être "plutôt vertical", en fonction de la pente du segment.

L'algorithme s'écrit de la façon suivante :

```
int dx, dy, i, xinc, yinc, cumul, x, y;
int xi, yi, xf, yf;
int rgb = 0xFF0000; // = "rouge"
x = xi;
y = yi;
dx = xf - xi; //On calcul la variation suivant x
dy = yf - yi; //Puis celle suivant y

//On affecte les valeurs adéquates pour xinc et yinc///////
if (dx > 0) {
    xinc = 1;
} else {
    xinc = -1;
}

if (dy > 0) {
    yinc = 1;
} else {
    yinc = -1;
}
//////////////////////////////////////////////////////////

dx = Math.abs(dx);
dy = Math.abs(dy);

monImage.setRGB(x, y, rgb); // On "allume" le pixel de coordonnées (x,y)
                            // le premier pixel éclairé est naturellement
                            // le point de départ.


// On allume un pixel en chaque abscisse entière x de xi à xf.

if (dx > dy) {
    cumul = dx / 2; // la variable cumul est utilisée pour stocker une
                    // estimation de la différence entre l'ordonnée
                    // entière et l'ordonnée réelle.

    for (i = 1; i <= dx; i++) {
        x += xinc; // On incrémente x de xinc
        cumul += dy;
        if (cumul >= dx) {
            cumul -= dx;
            y += yinc; // Puis y de yinc
        }

       monImage.setRGB(x, y, rgb);
```

```
        }
    } else { // Idem que la partie précédente dans le cas ou la variation
             // suivant y est plus grande que celle suivant x
        cumul = dy / 2;
        for (i = 1; i <= dy; i++) {
            y += yinc;
            cumul += dx;
            if (cumul >= dy) {
                cumul -= dy;
                x += xinc;
            }

             monImage.setRGB(x, y, rgb);
        }
    }

    //Puis on réinitialise l'objet porte pour détecter une nouvelle porte.
```

Figure 3 : Algorithme de Bresenham

Nous traçons ici un trait en coloriant en rouge un seul pixel à la fois. En agrandissant, on obtient une image avec des pixels disposés de telle sorte :

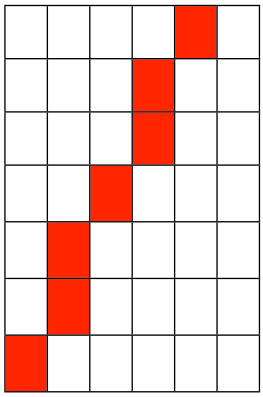

Figure 4 : Tracé de sortie primaire

Mais ce système ne marche pas avec l'algorithme de comportement que nous utilisons pour valider le simulateur.

Principe général de l'algorithme de comportement "basique" utilisé :

Pour éviter que les personnes se superposent, on a considéré au départ que chaque individu présent sur un pixel analysait la présence ou non d'individus sur les pixels situés sur un rayon 1 :

Considérons que la personne se trouve dans le pixel vert au centre.

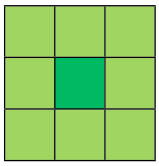

Figure 5 : Position de la personne avec un rayon 1

Si elle souhaite se déplacer vers le pixel du dessus, elle va regarder si les pixels autour du pixel visé (représenté en gris) sont libres. Si c'est le cas, elle va se déplacer.

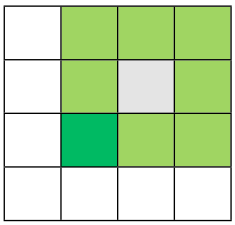

Figure 6 : Déplacement d'une personne avec un rayon 1

Pour améliorer le comportement des usagers de l'aéroport, nous avons choisi de réaliser le même principe, mais cette fois avec un rayon 2. Les personnes seront ainsi moins proches les unes des autres et suivront le même principe.

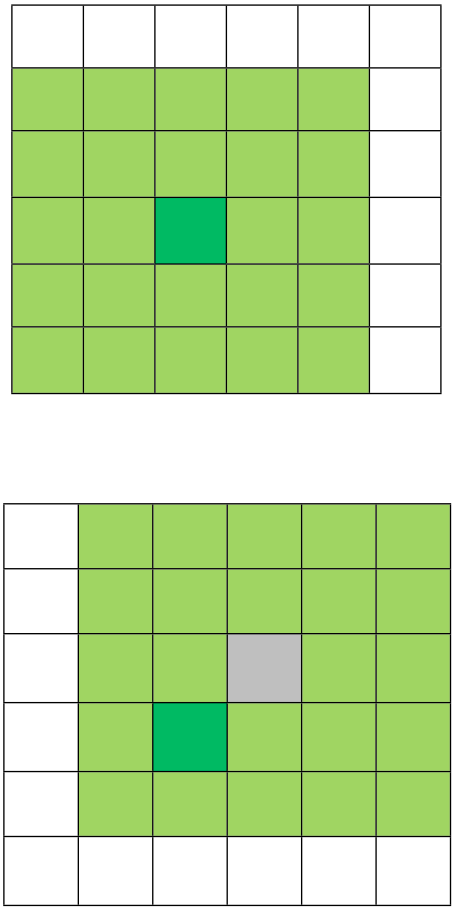

Figure 7 : Déplacement d'une personne avec un rayon 2

Dans le cas où il n'y a qu'un seul pixel rouge au niveau de la sortie et un rayon 2, le personnage ne va pas se déplacer car il détecte qu'il ne peut pas aller sur le pixel visé en gris à cause des pixels noirs.

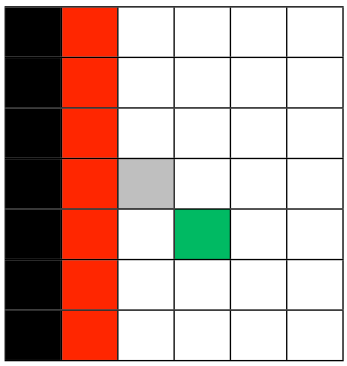

Figure 8 : Problème de déplacement avec un rayon 2 au niveau des sorties

Pour que les personnages puissent ainsi atteindre les sorties, il faut donc, quand on les détecte, rajouter des pixels rouges autour du pixel qui prend la couleur rouge avec l'algorithme de Bresenham. On obtient ainsi un tracé comme suit :

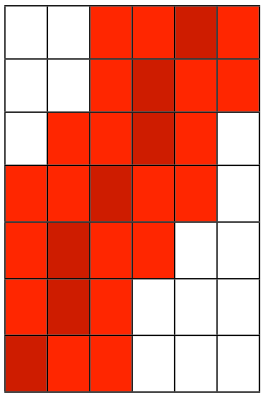

Figure 9 : Tracé final adapté

e) Finir la détection des portes

Pour pouvoir pleinement exploiter l'algorithme de Fast Marching, il faut encore réaliser une dernière étape. En effet, ce dernier se propage depuis la ou les sorties précédemment trouvées mais dans toutes les directions, il est donc nécessaire de transformer l'extérieur de la salle (zone verte de la ***figure 4***) de simulation en zone « obstacle ».

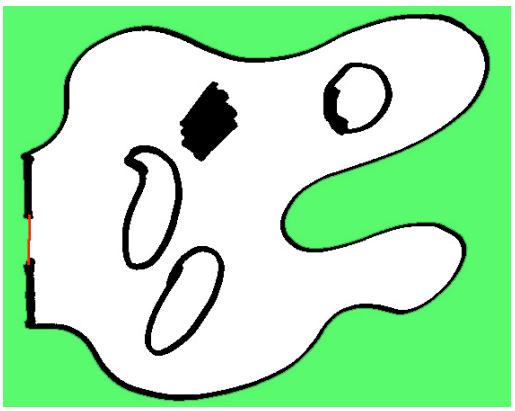

Figure 10 : Extérieur de la salle aéroportuaire à transformer en noir

Il s'agit donc de transformer les pixels blancs à l'extérieur de la zone en pixels noirs.

Pour cela nous avons appliqué l'**algorithme de FloodFill** (ou **« algorithme de remplissage par diffusion »)** que chacun a probablement déjà eu l'occasion d'utiliser sans le savoir au sein d'un logiciel de traitement d'image (le fameux « pot de peinture »).

Cet algorithme prend trois paramètres pour une image donnée :

- la position du pixel de départ (appelé aussi *germe*),

- la couleur ciblée (*colcible*, ici le blanc),

- la couleur de remplacement (*colrep*, ici le noir).

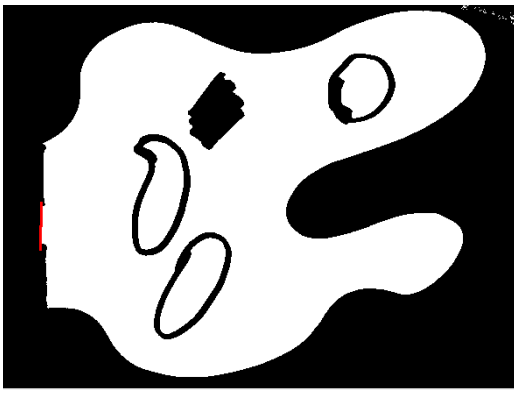

Figure 11 : Image traitée par l'algorithme de FloodFill

L'algorithme recense tous les pixels de l'image qui sont connectés au *germe* par un chemin de la couleur ciblée et substitue à cette dernière la couleur de remplacement. Il y a plusieurs manières de structurer cet algorithme mais celle que nous avons retenue est la suivante :

Soit P une pile vide

Si couleur(pixel considéré) ≠ colcible **alors**

sortir de l'algorithme

Fin si

Tant que P est non vide faire

Dépiler le *premier pixel n* de P

couleur(n) <- colrep

si couleur(*pixel au dessus de n*) = colcible **alors**

Empiler pixel au dessus de n sur P,

fin si

si couleur(*pixel au dessous de n*) = colcible **alors**

Empiler pixel au dessous de n sur P,

fin si

si couleur(*pixel à droite de n*)= colcible **alors**

Empiler pixel à droite de n sur P,

fin si

si couleur(*pixel à gauche de n*)= colcible **alors**

Empiler pixel à gauche de n sur P,

fin si

fin tant que

fin algorithme de FloodFill

3. Conclusion

À la suite de tout ces traitements, l'image comporte donc trois couleurs (blanc, noir et rouge). En faisant une simple association, on fournit au Fast Marching un équivalent (un tableau de **Point**) de l'image avec un codage spécifique pour chaque pixel (-2 pour les obstacles, -1 zone vierge et 1 pour les sorties).

B. Conception de l'IHM et mécanismes rattachés

1. Notre interface

Contrairement au projet ADA, nous avons ici fait le choix d'améliorer l'interaction entre le système et l'utilisateur à l'aide d'une interface graphique (en ADA, les dialogues avec l'utilisateur, pour la saisie de paramètres par exemple, était faits dans le terminal). Cette interface a été réalisée après une étude soignée des besoins (***cf Annexe n°2 : Récapitulatif des cas d'utilisation, des besoins et de leur criticité***), basée sur les éléments du cahier des charges énoncés par nos clients, et grâce à la bibliothèque graphique Swing.

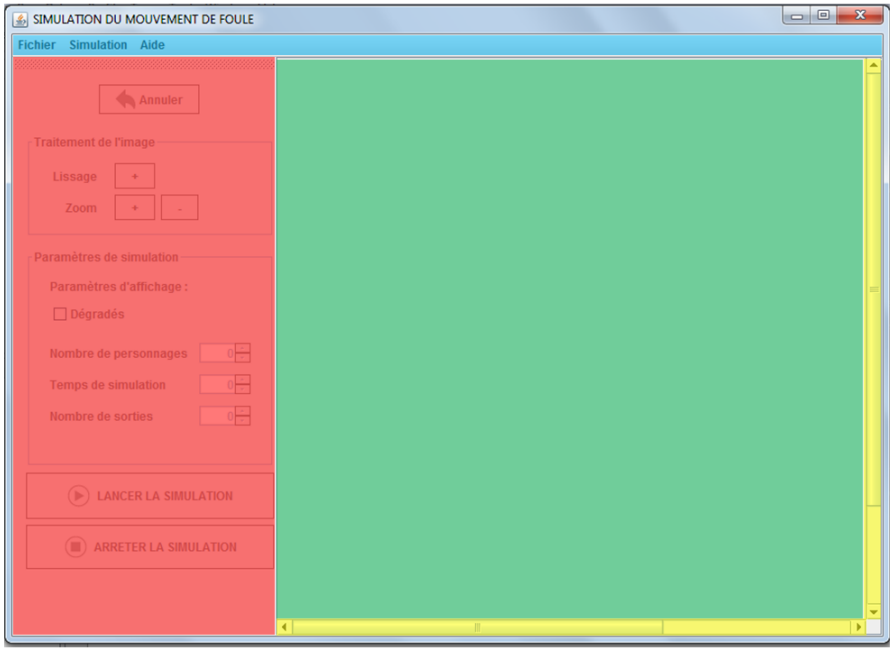

Figure 12 : Interface graphique

L'interface graphique se compose d'un grand container, une *JFrame*, qui contient elle même :

- en bleu une barre de menus, pour ouvrir des images, accéder au manuel utilisateur, lancer ou relancer une simulation,

- en rouge une toolbar qui contient tous les boutons pour le traitement de l'image et pour lancer et stopper la simulation,

- en jaune les scrollbars du *JScrollPanel* qui contient le *JPanel* en vert où l'on va afficher l'image et la simulation.

Choix des paramètres de simulation :

L'interface doit permettre à l'utilisateur de sélectionner ses paramètres de simulation. Pour cela, on utilise des *JSpinners* :

- un *Jspinner* appelé **jSpinner_nb_personnages** qui permet de renseigner le nombre de personnages de la simulation,

- un *Jspinner* appelé **jSpinner_nb_sorties** qui permet de renseigner le nombre de sorties présentes dans l'image,

- un *Jspinner* appelé **jSpinner_tps_simulation** qui permet de renseigner le temps total de la simulation.

Pour éviter toute interaction malencontreuse, les différents boutons et les spinners sont désactivés lorsque l'on ne doit pas les utiliser. À l'ouverture de l'application par exemple, tout est désactivé hormis la barre de menu qui permet d'accéder au menu d'ouverture d'une image.

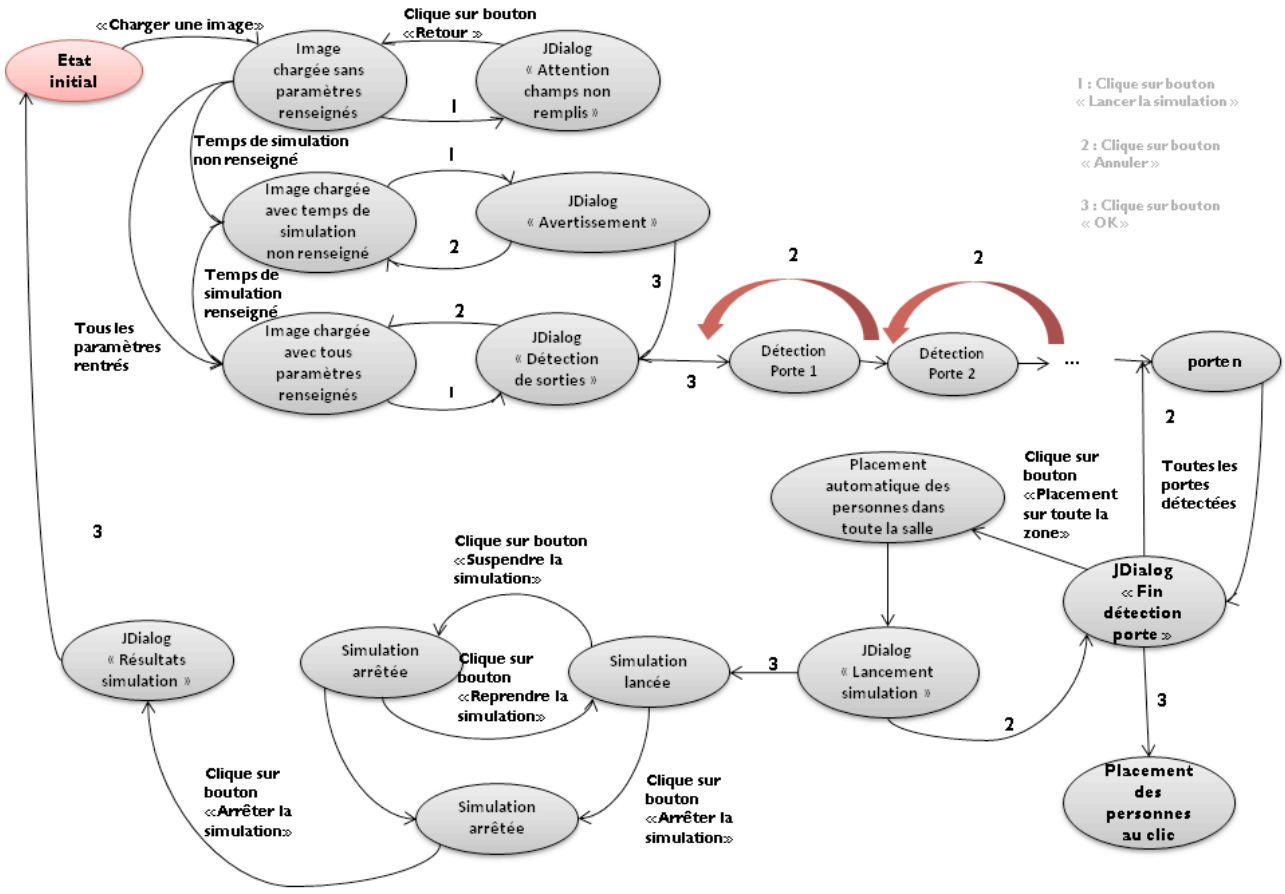
«Charger une image»
Etat initial
Image chargée sans paramètres renseignés
Clique sur bouton «Retour »
JDialog « Attention champs non remplis »
Temps de simulation non renseigné
Image chargée avec temps de simulation non renseigné
JDialog « Avertissement »
Temps de simulation renseigné
Tous les paramètres rentrés
Image chargée avec tous paramètres renseignés
JDialog « Détection de sorties »
Détection Porte 1
Détection Porte 2
...
porte n
1 : Clique sur bouton « Lancer la simulation »
2 : Clique sur bouton « Annuler »
3 : Clique sur bouton « OK »
Placement automatique des personnes dans toute la salle
Clique sur bouton «Placement sur toute la zone»
Toutes les portes détectées
JDialog « Fin détection porte »
Clique sur bouton «Suspendre la simulation»
JDialog « Résultats simulation »
Simulation arrêtée
Clique sur bouton «Reprendre la simulation»
Simulation lancée
JDialog « Lancement simulation »
Clique sur bouton «Arrêter la simulation»
Simulation arrêtée
Clique sur bouton «Arrêter la simulation»
Placement des personnes au clic

Figure 13 : Machine à états

2. Mécanismes particuliers

Fonction « annuler » :

Partant du constat que les actions de zoom sont destructices, il a fallu créer une fonction annuler. Pour cela nous avons créé deux piles pour stocker d'une part une copie de l'image avant chaque traitement et d'autre part un « souvenir » de l'opération passée :

- on stocke un 1 dans la pile lorsqu'on effectue un lissage.
- on stocke un 2 dans la pile des pour un zoom « plus ».
- on stocke un 3 dans la pile pour un zoom « moins ».
- on stocke un 4 dans la pile pour une action « annuler ».

En effet, si l'utilisateur effectue un zoom « plus » et qu'il clique ensuite sur zoom « moins », il ne faut pas effectuer un réel zoom « moins » mais plutôt annuler le zoom « plus ».

C. Algorithme de Fast Marching et descente de gradient

La descente de gradient effectuée lors du premier projet n'étant pas optimale, la première étape a d'abord consisté à l'améliorer en ADA. Après avoir constaté son optimisation, la traduction en elle même a pu être débutée.

La traduction du Fast Marching a résulté sur quelques modifications d'implémentation qui permettent d'exploiter au mieux le caractère objet du langage JAVA.
Les classes utilisées seront rassemblées dans un paquetage ***FastMarching***:

-***fm*** : classe contenant l'implémentation de l'algorithme de Fast Marching. Cet algorithme est mis en ouvre à l'aide d'une fonction appelée en interne par *CalculVoisin* qui permet de calculer un voisin en particulier donné par ses coordonnées,

-***Image*** (Interface faite mais non utilisée) : contenant les méthodes abstraites implémentées dans ImageMat.java,

-***ImageMat*** : classe qui construit une matrice image à partir de ses dimensions lignes et colonnes,

-***Point*** (dépendant de l'interface *Comparable*) : *Point* modélise un pixel de l'image dans lequel on veut garder certaines informations pour le Fast Marching telles que l'état(*Ombre*, *Pénombre*, *Eclaire*), la distance à la sortie (*val*) ou encore les coordonnées.

Les tests unitaires ont été élaborés au fur et à mesure de la traduction. Une fois la traduction faite, on testera la non régression par rapport au code ADA puis son intégration dans le reste du code.

D. Sauvegarde et récupération des données de simulation

La sauvegarde et la récupération des données des simulations ayant déjà été faites lors de la première phase du projet, elles faisaient partie de notre cahier des charges.

Une classe **Sauvegarde** a été créée, comprenant deux méthodes principales :

- une méthode ***écriture,***

- une méthode ***lecture***.

1. Sauvegarde

La méthode ***écriture*** écrit ligne à ligne dans un fichier les caractéristiques des points d'une *ImageMat* utilisée par l'algorithme de Fast Marching. On écrit ainsi :

- la valeur de la hauteur de l'image sur la première ligne du fichier,

- la valeur de la longueur de l'image sur la deuxième ligne du fichier,

- pour chaque point ses coordonnées entières suivies de sa valeur, qui représente la distance à la sortie. Chaque caractéristique est séparée par un espace et le passage à la ligne se fait quand on a atteint les caractéristiques du Point situé le plus à gauche de l'image.

Pour cela, nous utilisons la classe *BufferedWriter*(java.io), construit à partir d'un *FileWriter*(java.io). On peut ainsi écrire ligne à ligne et non pas caractère par caractère.

2. Récupération

Inversement, il faut pouvoir récupérer les données d'une simulation sauvegardée. La méthode ***lecture*** lit directement les données de simulation écrites dans un fichier.

On utilise la classe *Scanner* (java.util), construit à partir d'un *FileReader*(java.io). Cette fois les caractères sont lus trois par trois pour récupérer les caractéristiques d'un même point.

Les valeurs récupérées sont ensuite utilisées pour construire des Points utilisables par l'algorithme de Fast Marching, qui sont ensuite stockés dans un tableau de Points.

Remarque : Cette méthode de sauvegarde a été implémentée mais n'est pas utilisée car nous lui avons préféré la «sérialisation» (qui est en cours de réalisation). Cette méthode prend des données d'objets en mémoire, les transforme en séquence d'octets pour les stocker dans des fichiers.

III. Implémentation

A. Mise en oeuvre de l'algorithme de comportement Uzawa

Le modèle de mouvement de foule que nous proposons repose sur deux principes. D'une part, on définit un champ de vitesses souhaitées, correspondant au déplacement que ferait une personne seule. D'autre part, le déplacement effectif doit respecter une contrainte d'encombrement maximal (**Cf. Annexe 1: Aspects mathématiques de l'algorithme d'Uzawa**).
Les N individus sont assimilés à des disques de rayon r, repérés par les coordonnées de leurs centres:

$$q = (q_1, .., q_N) \in \mathbb{R}^{2N}$$

Les obstacles sont représentés par un ensemble de segments. Par souci de clarté, on ne présente que la gestion des contacts entre personnes mais il en est de même des contacts entre les personnes et les obstacles.

Les disques ne devant pas se chevaucher, le vecteur position q doit appartenir à un ensemble de configurations admissibles:

$$Q_0 = \{q \in \mathbb{R}^{2N}, \forall i, j \quad D_{ij}(q) = \|q_i - q_j\| - 2r \geqslant 0\}$$

Figure 14 : Représentation de la distance entre deux personnes i et j

Remarque:

- Nous avions initialement prévu d'utiliser la classe Uzawa déjà existante, mais celle-ci n'étant pas assez documentée, nous avons préféré l'implémenter par nous même.
- l'algorithme d'Uzawa ne fonctionne pas lorsque qu'il n'y a qu'une seule personne dans la zone à étudier (IndexOutOfBounds).

B. Mise en oeuvre de l'algorithme d'itération Euler

Cet algorithme va permettre de faire progresser les personnes dans la salle au cours de la simulation. Notre algorithme d'itération consiste à récupérer le vecteur vitesse (vitesse réelle de l'individu) et à le multiplier par le pas de temps pour mettre à jour sa position de départ (***cf. Figure 9: Synthèse de l'utilisation des algorithmes***). La classe ***Euler*** initialement implémentée ne sera plus utilisée, en effet, les itérations sont directement calculées dans la classe ***Uzawa***.

C. Synthèse

Les algorithmes mis en oeuvre précédemment s'enchainent de la manière suivante:

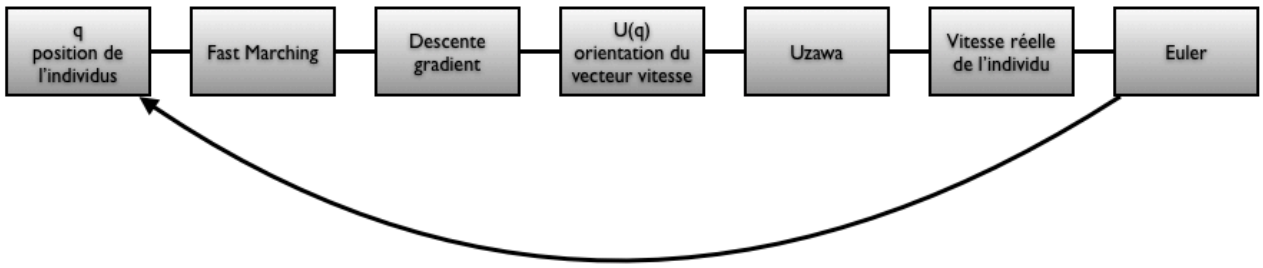

Figure 15 : Synthèse de l'utilisation des algorithmes

IV. Organisation et méthodes

A. Division du travail

Grâce au management de projet, à l'analyse du besoin et la conception orientée objet qui nous a permis de mettre en place le modèle du domaine et la définition des cas d'utilisation, nous avons pu nous répartir les tâches à effectuer en deux parties :

-Cédric et Estelle se sont chargés de la traduction de l'algorithme de Fast Marching et de la descente de gradient, ainsi que de l'implémentation de l'algorithme d'Uzawa,

-Maxime et Margaux, se sont quant à eux occupés de la traduction du traitement d'image ainsi que de l'implémentation de l'interface de notre simulateur et à la sauvegarde des données.

Margaux a été nommée chef de projet et s'est donc chargée de la coordination des différentes phases de notre projet, du bon suivi de notre planning, des compte-rendus d'avancement.

Nous avons ainsi pu réaliser les premières recherches sur nos parties et au fur-et-à-mesure de l'avancée de la conception objet, nous avons pu commencer à coder. Le but était de traduire rapidement ce qui avait été déjà implémenté pour se pencher au plus vite sur les nouveautés du cahier des charges.

En parallèle, nous devions avancer le plan de projet, le dossier des besoins, le dossier de conception orientée objet, le rapport de synthèse et le plan de vérification et validation.

Le regroupement de nos deux parties a débuté début juin. Par manque d'heures de projet programmées, et au vu du retard pris dans la compréhension de l'algorithme d'Uzawa s'avérant plus complexe que prévu, le regroupement a pris un peu de retard sur notre planification. La qualité de nos développements et leur conformité avec la conception nous a permis de regrouper rapidement le code et d'obtenir de bons résultats sur le programme commun.

Les précautions et l'organisation prises dès le début du projet nous ont donc très fortement aidé et permis d'avancer très rapidement dans un processus où nous pensions passer de nombreuses heures. Finalement, après avoir mis nos parties en commun, nous avons aussi pu passer plus de temps sur les parties de management de projet (rapports, dossier de tests, etc).

B. Suivi du projet

Conformément à la planification élaborée dans notre plan de projet, l'avancement a été globalement respecté chaque semaine. Un rapport d'avancement a également été envoyé à nos tuteur tous les quinze jours environ.

De plus, des réunions d'avancement réparties sur la période impartie nous ont permis de tenir au courant nos tuteurs du bon déroulement et de l'avancée de notre projet.

Notre projet a pris un peu de retard au début dû à l'absence d'heures entièrement dédiées au projet, à l'assimilation tardive des cours de conception orientée objet, des cours JAVA et des TP JAVA n'ayant finis que le 27 mai, il a donc fallu élaborer notre phase de développement en dehors de nos heures de cours et ainsi anticiper certaines parties de notre implémentation, à savoir par exemple l'élaboration de notre interface.

C. Contraintes

Les contraintes d'organisation ne sont pas nombreuses mais jouent néanmoins un rôle important dans le suivi de notre projet, notamment les logiciels de partage de fichiers (Dropbox) et de codes sources (GIT).

- GIT est un système de management de code source et fait parti de nos livrables. En effet, le rendu des différents rapports ainsi que des codes sources doit se faire sur cette plateforme. Cependant, nous avons rencontré quelques difficultés lors de son installation nécessaire mais néanmoins complexe et son adaptation, dues à nos différents systèmes d'exploitation (Windows, Linux, Mac OS). Le risque concernant GIT est contenu dans celui lié au temps d'adaptation logiciel, qui avait été défini dans le plan de projet, (partie 10.b). Nous sommes néanmoins conscients que c'est un outil utilisé dans la vie professionnelle des ingénieurs, et qu'il peut être plus efficace pour travailler en groupe que d'autres techniques moins poussées – mais nous pensons que globalement cet outil a créé plus de problèmes et nous a fait perdre du temps.

- Dropbox est un service de stockage et de partage de copies de fichiers locaux en ligne. Nous l'utilisons pour partager et modifier nos fichiers, même les fichiers code source. Cependant pour que ces modifications soient visibles par tous, il faut s'assurer que le document ne soit ouvert qu'une seule fois durant l'enregistrement de ces modifications, et non sur plusieurs postes à la fois.
  Si plusieurs personnes travaillent dessus en même temps ou que le document est encore ouvert sur un autre poste, les modifications seront visibles uniquement par la personne qui les a effectué, sur son compte. Ses collaborateurs garderont la version non modifiée.
  Il faudra donc veiller à enregistrer les nouvelles versions sous un nouveau fichier afin de repérer la version originale sur son compte.

V. Retour d'expérience

A. Mise en oeuvre de l'architecture

La phase de conception objet a été grandement facilitée par notre connaissance du cahier des charges de la première phase du projet. Cette analyse a été tout de même modifiée car aux besoins de la phase 1 se sont rajoutés les nouveaux besoins de notre nouveau cahier des charges.

La conception objet faite au début de la phase 2, a été très fiable et de bonne qualité. En effet, nous n'avons pratiquement pas modifié celle-ci et qui plus est, les résultats, en termes de performance et de qualité ont été satisfaisants. Les seules modifications de conception qui ont été faites concernent le modèle du domaine :

-la classe ***AlgoBasique+*** (reprise de ***AlgoBasique*** en multi-thread) a été supprimée car le multi-thread ici n'est pas utile.

-les classes ***PixelObstacle***, ***PixelSortie***, ***PixelZoneVierge*** dérivées de la classe ***Pixel*** ont été supprimées au profit de la classe ***Point***, représentation d'un ***Pixel*** de ***ImageMatrice***. La différenciation des différents pixels est faite à l'aide de coefficients pour chaque type de points.

-la classe ***ImageMatrice*** a été rajoutée pour représenter la ***ZoneAéroportuaire*** et est constituée des différents points.

L'interface développée initialement a été améliorée. En effet, les calculs sont maintenant effectués avant l'affichage de la simulation pour permettre à l'utilisateur d'adapter la vitesse de visionnage de la simulation.

B. Tests

Les tests effectués tout au long de l'implémentation de notre simulateur ont permis de valider les éléments du cahier des charges soumis : (cf.I.B. Cahier des charges)

-la traduction en JAVA du logiciel développé en ADA a été effectuée dans son intégralité. Les fonctionnalités ont pu même être améliorées:

-performances : les temps de calcul ont été constatés plus courts en JAVA,

-le placement aléatoire des personnes se fait dans une zone choisie par l'utilisateur.

- l'algorithme de comportement ayant servi à la validation du logiciel en phase 1 a été mis en ouvre et également amélioré ***(cf.II.A.2.d) Détection de la (des) porte(s))***

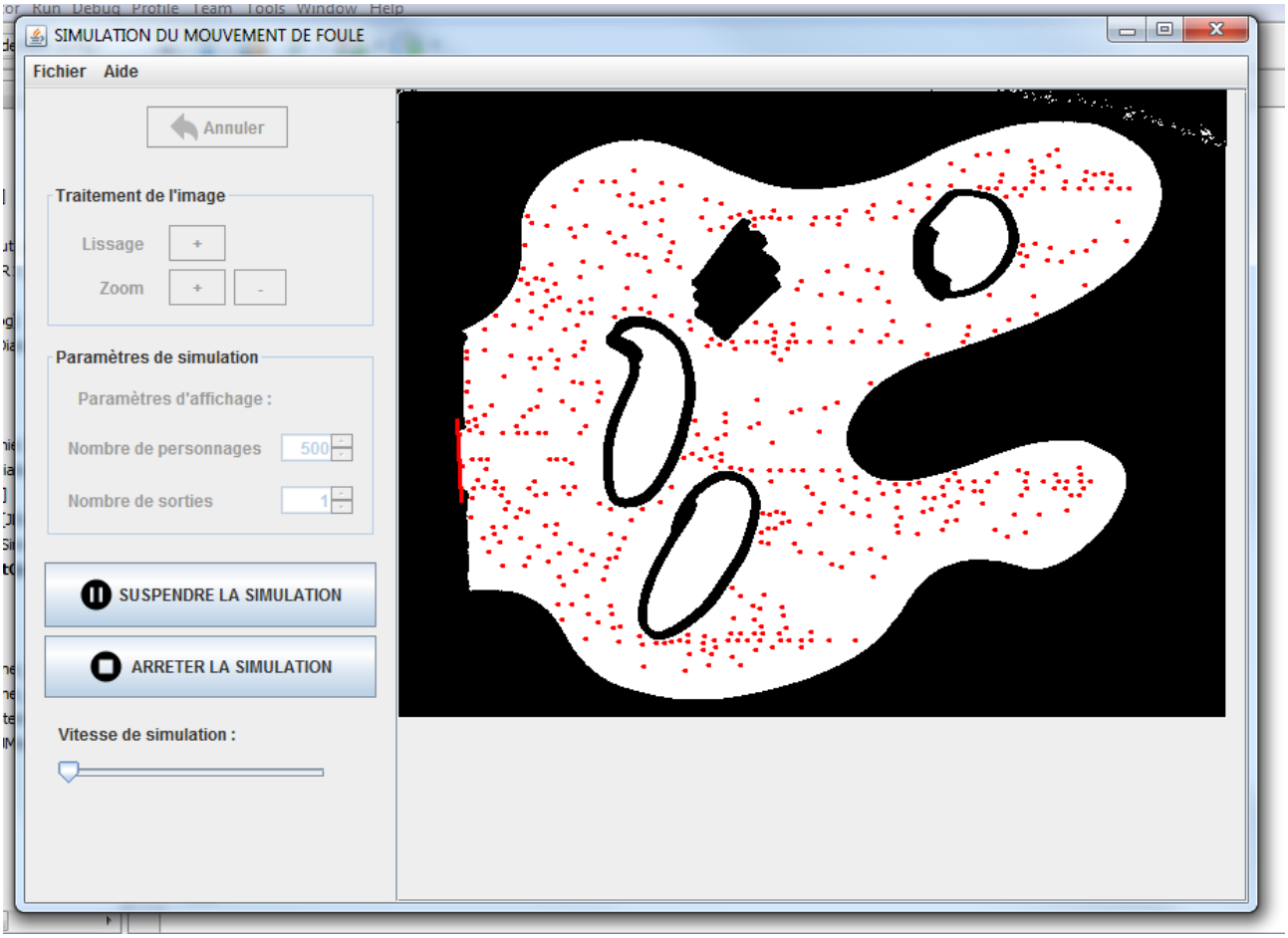

Figure 16 : Lancement d'une simulation avec l'algorithme de comportement basique

- l'algorithme d'évolution d'une foule, à savoir l'algorithme d'Uzawa, a été implémenté, cependant on contaste qu'il ne converge pas de façon à chaque fois de manière certaine.

C. Matrice de conformité

(cf Annexe 2: Récapitulatif des cas d'utilisation, des besoins et de leur niveau de priorité)

Nous avons choisi de garder ici les besoins principaux, c'est à dire ceux correspondant au niveau de priorité 1.

Identifiant	Besoin	Solution
B1	*Fonctions*	
B1.1	Permettre à l'utilisateur de visualiser une simulation.	Implémentation d'une interface
B1.1.2	Permettre aux utilisateurs de visualiser **sans interruption** une simulation de mouvements de foule dans une zone aéroportuaire.	Dans l'interface, implémentation d'une **toolbar** contenant un bouton «Lancer la simulation»
B1.1.3	Permettre aux utilisateurs de visualiser **avec interruption** une simulation de mouvements de foule dans une zone aéroportuaire.	Dans l'interface, implémentation d'une **toolbar** contenant un bouton «Arrêter la simulation»
B1.1.4	Permettre aux utilisateurs de visualiser une simulation sauvegardée.	Interface + implémentation d'un algorithme de sauvegarde
B1.2	Permettre aux utilisateurs de sauvegarder les données d'une simulation interrompue dans un système de gestion de données adapté.	Interface + implémentation d'un algorithme de sauvegarde
B1.3	Le système permet à l'utilisateur de choisir les paramètres d'utilisation du simulateur (zoom, nombre de personnages, zone aéroportuaire à traiter...).	Dans l'interface, implémentation d'une **toolbar** utilisant des Jspinners
B1.5	Le système utilise les algorithmes de Fast Marching et d'Uzawa pour permettre à l'utilisateur de visualiser le comportement d'une foule dans une zone aéroportuaire.	Implémentation des algorithmes de Fast Marching et d'Uzawa
B2	*Performances attendues*	

Identifiant	Besoin	Solution
B2.1	Les temps de calcul doivent être équivalents ou plus faibles que ceux obtenus en ADA.	Affichage des résultats et des simulations en JAVA pour les comparer à ceux obtenus en ADA
B3	*Interfaces*	
B3.1	Le système doit pouvoir récupérer l'image photographiée par l'utilisateur.	"ouvrir une image" depuis l'interface
B3.1.1	Le système doit traiter l'image (binariser et lisser (selon la volonté de l'utilisateur)).	Propriété de la toolbar (cf II.A.Traitement d'image) + binarisation faite au chargement de l'image
B3.1.2	Le système doit permettre d'afficher une image de la simulation.	Propriété de l'interface
B4	*Scénarios opérationnels*	
B4.1	Le système permet de réaliser le cas d'utilisation "CU1 - Lancement d'une nouvelle simulation".	Dans l'interface, implémentation d'une **toolbar** contenant un bouton «Lancer la simulation»
B4.2	Le système permet de réaliser le cas d'utilisation "CU2 - Lancement d'une simulation sauvegardée".	Interface + implémentation d'un algorithme de sauvegarde
B4.4	Le système permet de réaliser le cas d'utilisation "CU4 - Interrompre la simulation".	Dans l'interface, implémentation d'une **toolbar** contenant un bouton «Arrêter la simulation»
B4.5	Le système permet de réaliser le cas d'utilisation "CU5 - Reprendre la simulation".	Dans l'interface, implémentation d'une **toolbar** contenant un bouton «Reprendre la simulation»

Identifiant	Besoin	Solution
B4.6	Le système permet de réaliser le cas d'utilisation "CU6 - Enregistrement de la simulation".	Interface + implémentation d'un algorithme de sauvegarde
B4.7	Le système permet de réaliser le cas d'utilisation "CU7 - Charger une image".	menu interface
B4.8	Le système permet de réaliser le cas d'utilisation "CU8 - Rentrer les paramètres".	Jspinners et toolbar
B4.10	Le système permet de réaliser le cas d'utilisation "CU10 - Placer les personnes".	Dans l'interface, implémentation d'une **toolbar** utilisant des Jspinners + placement des personnes à la main
B5	*Contraintes*	
B5.1	Le système doit enregistrer les données de chaque simulation à intervalles de temps réguliers.	implémentation d'un algorithme de sauvegarde
B5.3	Le système doit conserver l'ensemble des fonctionnalités du simulateur réalisé en ADA.	Traduction de l'ensemble des fonctionalités du code ADA en code JAVA
B5.4	Le système doit offrir à l'utilisateur un paramétrage suffisant.	propriété interface
B5.5	Le système doit être accompagné d'un manuel d'utilisation.	Rédaction d'un manuel d'utilisation utilisateur

VI. Conclusion

Ce deuxième projet de simulation de mouvement de foule nous a tout d'abord permis d'appliquer nos connaissances en langage objet, qu'il s'agisse de la conception, ou de l'implémentation en JAVA. En plus du développement, nous avons mené un management de projet efficace et rigoureux, conçu au début du projet, et suivi tout au long de celui-ci. Notre méthodologie générale a d'ailleurs porté ses fruits, puisque le projet a été terminé dans les délais impartis.

Lors de cette deuxième phase de projet, nous avons pu redévelopper un simulateur de mouvements de foule en JAVA. À partir de la première phase déjà réalisée, à l'aide de la définition des besoins et de la conception orientée objet, nous avons pu traduire et améliorer en JAVA les algorithmes principaux déjà implémentés en ADA (algorithme de Fast Marching, traitement d'image, algorithme de comportement basique). L'algorithme de comportement Uzawa a ensuite été mis en oeuvre ainsi qu'une interface entre le système et l'utilisateur.

Cet algorithme de comportement pourra être éventuellement amélioré avec l'introduction d'une distance au-delà de laquelle l'intéraction entre les personnes ne sera pas traitée.
Les résultats obtenus lors de ce projet pourront servir à d'autres projets informatiques pour continuer à simuler des mouvements de foule en fonction d'autres algorithmes de comportement.

VII.Annexes

Annexe 1 : Aspects mathématiques de l'algorithme d'Uzawa

Contexte général

L'algorithme d'Uzawa est utilisé pour trouver le minimum d'une fonction $J(x)$ sous certaines contraintes. Plus précisément, pour résoudre le problème suivant :

$$(P)\begin{cases}\min_{x\in C} J(x),\\ C = \{x \in \mathbb{R}^n : \varphi^i(x) \leq 0, i = 1, \dots, p\}.\end{cases}$$

La fonction J est une fonction de coût et les contraintes sont linéaires. La formulation de ce problème s'obtient par une formulation duale qui permet d'intégrer les contraintes dans une nouvelle fonction de coût. Pour cela, on introduit le Lagrangien, qui est une fonction L définie sur $\mathbb{R}^n \times \mathbb{R}^p_+$ par :

$$\begin{aligned} L(x,\mu) &= J(x) + \sum_{i=1}^{p} \mu_i \varphi^i(x) \\ &= J(x) + \langle \mu, \varphi(x) \rangle, \end{aligned}$$

avec $\varphi := (\varphi^1, ..., \varphi^p)$ et $\mu \in \mathbb{R}^p_+$ le vecteur des paramètres de Lagrange.

Dans la suite, on supposera l'existence d'un point selle de $L(x,\mu)$, i.e un point (x_*, μ_*) tel que :

$$\max_{\mu\geq 0} L(x_*,\mu) = L(x_*,\mu_*) = min_x L(X,\mu_*)$$

$$L(x_*,\mu_*) = \max_{\mu\geq 0} \min_{x\in\mathbb{R}^n} L(x,\mu).$$

Si μ_* connu, alors le problème (P) devient un problème sans contraintes :

$$(P_{\mu_*}) \quad \min_x L(x,\mu_*).$$

Comment trouver μ_* ? Pour cela, introduisons la fonction G définie sur $\mathbb{R}^p$ par

$$G(\mu) = \inf_x L(x,\mu) = L(x_*,\mu),$$

et notons, qu'alors $\mu_* = \arg\max_{\mu\geq 0} G(\mu)$. Introduisons également le projecteur

$$\begin{aligned} \Pi_+ : \mathbb{R}^p &\to \mathbb{R}^p_+ \\ (\mu_1, ..., \mu_p) &\mapsto (\max(\mu_1, 0), ..., \max(\mu_p, 0)) \end{aligned}$$

Pour obtenir μ_*, on utilise un algorithme du type gradient projeté,

$$\mu_{k+1} = \Pi_+(\mu_k + \rho \nabla G(\mu_k))$$

où ∇G est le gradient de G, soit encore

$$\nabla G(\mu) = (\varphi^1(x_*), ..., \varphi^p(x_*)).$$

L'algorithme d'Uzawa

L'algorithme d'Uzawa se compose de deux étapes, la première consistant à minimiser le Lagrangien par rapport à x, puis à proposer un paramètre de Lagrange par une méthode de gradient projeté. Pour la convergence de l'algorithme, il faut que ρ ne soit pas trop grand.

Algorithm 1: Algorithme d'Uzawa

Entrées: $J, \varphi^i, i = 1, \dots, p, \rho, \epsilon$
Sorties: x, μ
begin
  $\mu_0 \longleftarrow 0$;
  répéter
    Calculer $x_k = \arg\min L(x,\mu_k)$;
    $\mu_{k+1} \longleftarrow \Pi_+(\mu_k + \rho \nabla G(\mu_k))$
  jusqu'à $\|x_{k+1} - x_k\| \leq \epsilon$;

Exemple : coût quadratique

Étudions le cas simple d'un coût quadratique avec des contraintes linéaires. Donc la fonction de coût s'exprime par

$$J(x) = \frac{1}{2}x^T Ax - x^T b$$

avec $x \in \mathbb{R}^n$, A une matrice carrée $n \times n$ symétrique strictement définie positive et b un vecteur de $\mathbb{R}^n$. Les contraintes sont de la forme

$$Cx \leq d$$

avec C une matrice $p \times n$ et $d \in \mathbb{R}^p$. Notons que dans ce cas les fonctions φ^i sont données par

$$\varphi^i(x) = \sum_j C_{ij}x_j - d_i.$$

Par ailleurs, le Lagrangien s'exprime par

$$L(x, \mu) = \frac{1}{2}x^T Ax - x^T(b - C^T\mu) - \mu^T d$$

Comme le gradient de L par rapport à x s'exprime par

$$\nabla_x L = Ax - (b - C^T\mu),$$

la première étape d'Uzawa correspond au calcul de

$$x_k = A^{-1}(b - C^T\mu_k).$$

Pour la seconde étape, nous avons

$$G(\mu_k) = -\frac{1}{2}\mu_k^T(CA^{-1}C^T)\mu_k + \mu^T(CA^{-1}b - d)$$
$$\nabla G(\mu_k) = C[A^{-1}(b - C^T\mu) - d] = Cx_k - d,$$

d'où,

$$\mu_{k+1} = \Pi_+(\mu_k + \rho(Cx - d)).$$

Dans cet exemple, la valeur maximale de ρ est connue, elle s'exprime par :

$$\rho_{\max} = \frac{2\lambda_1}{\|C\|^2}$$

avec λ_1, la plus petite valeur propre de A et la norme $\|C\|$ de C étant égale à la plus grande valeur singulière de C (soit encore la plus grande valeur propre de la matrice carrée CC^T).

L'algorithme d'Uzawa dans ce cadre devient

Algorithm 2: Algorithme d'Uzawa pour le cas quadratique

Entrées: A, b, C, d, ρ, ϵ
Sorties: x, μ
begin
  $\mu_0 \longleftarrow 0$;
  $\rho_{\max} = \frac{2\lambda_1}{\|C\|^2}$;
  si $\rho > \rho_{\max}$ **alors**
    Sortir
  $x = A^{-1}(b - C^T\mu_0)$;
  $\mu = \Pi_+(\mu_0 + \rho(Cx - d))$;
  répéter
    $x_0 \longleftarrow x$;
    $x = A^{-1}(b - C^T\mu)$;
    $\mu \longleftarrow \Pi_+(\mu + \rho(Cx - d))$
  jusqu'à $\|x - x_0\| \leq \epsilon$;

Application au mouvement de foule

Nous avons en prenant les notations de la thèse de Venel

$$J(v) = \frac{1}{2}\|v - L(q)\|^2 = \frac{1}{2}\langle v, v\rangle - \langle v, U(q)\rangle + \text{cste}.$$

Donc, $A = Id$ et $b = U(q)$. Les contraintes sont

$$\forall i < j, \quad D_{ij}(q) + hG_{ij}(q)v \geq 0$$

où ici G_{ij} désigne le gradient de D_{ij}.

Introduisons la matrice G de taille $N(N-1)/2 \times 2N$, où N est le nombre d'individus, et le vecteur D de taille $N(N-1)/2$ définis respectivement par

$$G(q) = (G_{ij}(q)), \qquad D(q) = (D_{ij}(q)),$$

alors la contrainte s'exprime par

$$-hG(q).v \leq D(q).$$

On se retrouve donc avec le cas précédent où :

$$A = Id, \quad b = U(q), \quad C = -hG(q), \quad d = D(q).$$

Notons également que $\lambda_1 = 1$.

Annexe 2 : Récapitulatif des cas d'utilisation, des besoins et de leur niveau de priorité

Le tableau ci-dessous reprend les différents cas d'utilisation décrits dans la ***Tableau 1***, en donnant une description orientée "but pour l'utilisateur".

Cas d'utilisation	Description
CU1 - Lancement d'une nouvelle simulation.	Permettre à l'utilisateur de lancer une nouvelle simulation
CU2 - Lancement d'une simulation sauvegardée.	Permettre à l'utilisateur de lancer une simulation déjà commencée et donc sauvegardée.
CU3 - Comparaison des résultats.	Permettre à l'utilisateur de comparer les résultats de plusieurs simulations.
CU4 - Interrompre la simulation.	Permettre à l'utilisateur d'interrompre la simulation à tout moment.
CU5 - Reprendre la simulation.	Permettre à l'utilisateur de reprendre la simulation interrompue à tout moment.
CU6 - Enregistrement de la simulation.	Permettre à l'utilisateur de sauvegarder la simulation en cours.
CU7 - Charger une image.	Permettre à l'utilisateur de charger et donc d'afficher une image enregistrée.
CU8 - Rentrer les paramètres.	Permettre à l'utilisateur de choisir les paramètres de simulation.
CU9 - Sélectionner les sorties.	Permettre à l'utilisateur de placer où il le souhaite les sorties.
CU10 - Placer les personnes.	Permettre à l'utilisateur de placer aléatoirement les personnes.

Tableau 1 : Différents cas d'utilisation du système avec leur description

Les besoins sont présentés dans le ***tableau 3*** ci-dessous, en prenant la classification suivante :

- Besoins de type *Fonctions*;
- Besoins de type *Performances attendues*;
- Besoins de type *Interfaces*;
- Besoins issus des *Scénarios opérationnels*;
- Besoins de type *Contraintes*.

Ce ***tableau 3*** regroupe tous les besoins issus des points forts et des points faibles du simulateur de mouvements de foule réalisé en langage ADA (phase 1). Les différents besoins sont regroupés par niveaux de priorité, comme le montre le ***tableau 2.***

Niveau de priorité	Correspondance
Important	1
Intermédiaire	2
Secondaire	3

Tableau 2 : Niveaux de priorité

Identifiant	Besoin	Niveau de priorité
B1	*Fonctions*	
B1.1	Permettre à l'utilisateur de visualiser une simulation.	**1**
B1.1.2	Permettre aux utilisateurs de visualiser **sans interruption** une simulation de mouvements de foule dans une zone aéroportuaire.	**1**
B1.1.3	Permettre aux utilisateurs de visualiser **avec interruption** une simulation de mouvements de foule dans une zone aéroportuaire.	**1**
B1.1.4	Permettre aux utilisateurs de visualiser une simulation sauvegardée.	**1**
B1.1.5	Le système permet à l'utilisateur de revoir rapidement et facilement une simulation sauvegardée.	**2**
B1.2	Permettre aux utilisateurs de sauvegarder les données d'une simulation interrompue dans un système de gestion de données adapté.	**1**
B1.3	Le système permet à l'utilisateur de choisir les paramètres d'utilisation du simulateur (zoom, nombre de personnages, zone aéroportuaire à traiter...).	**1**
B1.4	Le système connaît les coordonnées des points en haut à gauche et en bas à droite de l'image.	**3**
B1.5	Le système utilise les algorithmes de Fast Marching et d'Uzawa pour permettre à l'utilisateur de visualiser le comportement d'une foule dans une zone aéroportuaire.	**1**
B2	*Performances attendues*	
B2.1	Les temps de calcul doivent être équivalents ou plus faibles que ceux obtenus en ADA.	**1**

Identifiant	Besoin	Niveau de priorité
B2.2	Le système n'arrête pas son exécution si l'utilisateur se trompe dans la gestion des paramètres.	3
B3	*Interfaces*	
B3.1	Le système doit pouvoir récupérer l'image photographiée par l'utilisateur.	1
B3.1.1	Le système doit traiter l'image (binariser et lisser).	1
B3.1.2	Le système doit permettre d'afficher une image de la simulation.	1
B3.2	Le système permet à l'utilisateur de lancer facilement la simulation.	2
B3.3	Le système signale une erreur en cas d'erreur de saisie d'un paramètre.	2
B3.4	Le système permet d'annuler la dernière action effectuée.	2
B3.5	Le système permet à l'utilisateur de savoir quel zoom est à appliquer pour une image en particulier.	3
B3.6	Le système permet à l'utilisateur de visualiser à la fois l'image traitée et l'évolution de la simulation.	2
B4	*Scénarios opérationnels*	
B4.1	Le système permet de réaliser le cas d'utilisation "CU1 - Lancement d'une nouvelle simulation".	1
B4.2	Le système permet de réaliser le cas d'utilisation "CU2 - Lancement d'une simulation sauvegardée".	1
B4.3	Le système permet de réaliser le cas d'utilisation "CU3 - Comparaison des résultats".	3
B4.4	Le système permet de réaliser le cas d'utilisation "CU4 - Interrompre la simulation".	1
B4.5	Le système permet de réaliser le cas d'utilisation "CU5 - Reprendre la simulation".	1
B4.6	Le système permet de réaliser le cas d'utilisation "CU6 - Enregistrement de la simulation".	1
B4.7	Le système permet de réaliser le cas d'utilisation "CU7 - Charger une image".	1
B4.8	Le système permet de réaliser le cas d'utilisation "CU8 - Rentrer les paramètres".	1
B4.9	Le système permet de réaliser le cas d'utilisation "CU9 - Sélectionner les sorties".	2
B4.10	Le système permet de réaliser le cas d'utilisation "CU10 - Placer les personnes".	1
B5	*Contraintes*	

Identifiant	Besoin	Niveau de priorité
B5.1	Le système doit enregistrer les données de chaque simulation à intervalles de temps réguliers.	**1**
B5.2	Le système doit offrir un temps de recherche de simulations sauvegardées adéquat.	**3**
B5.3	Le système doit conserver l'ensemble des fonctionnalités du simulateur réalisé en ADA.	**1**
B5.4	Le système doit offrir à l'utilisateur un paramétrage suffisant.	**1**
B5.5	Le système doit être accompagné d'un manuel d'utilisation.	**1**
B5.6	Le système fonctionne sur n'importe quel système d'exploitation (Linux, Windows, MacOS...).	**2**
B5.7	Le système n'arrête pas son exécution si l'utilisateur se trompe dans la gestion des paramètres.	**3**

Tableau 3 : Synthèse des besoins du simulateur

VIII.Bibliographie

A. Algorithmes de comportement et théorie des mouvements de foule

1) *Modélisation macroscopique de mouvements de foule*, A. Roudneff-Chupin, Thèse de doctorat d'université en mathématiques, faculté des sciences d'Orsay, université Paris XI – Paris-Sud, Paris, 12 décembre 2011.

2) *Simulation de mouvements de foules*, M. Azéma, M. André, C. Zimmermann et E. Boudassou, rapport de projet ADA/IA, IENAC 12S, ENAC, Toulouse, 7 février 2014.

3) *Un modèle macroscopique de mouvements de foule*, A. Roudneff-Chupin et B. Maury, Séminaire des doctorants d'Orsay, université Paris-Sud – Paris XI, Paris, 7 octobre 2010.

6) *Etude expérimentale et modélisation des déplacements collectifs de piétons*, M. Moussaïd, Thèse de doctorat d'université en éthologie, université de Toulouse III – Paul-Sabatier, Toulouse, 18 juin 2010.

10) *Modélisation de l'évacuation d'une foule*, N. Chabriac, Mémoire de TIPE (spécialité mathématiques), Lycée Pierre-de-Fermat, Toulouse, 10 avril 2012.

B. Algorithme d'Uzawa

11)*JAMA : A Java Matrix Package*
http://math.nist.gov/javanumerics/jama/

12) *Modélisation mathématique et numérique de mouvements de foule*, J. Venel, Thèse de doctorat d'université en mathématiques (n° d'ordre : 9245), faculté des sciences d'Orsay, université Paris-Sud – Paris XI, Paris, 27 novembre 2008.

C. Traitement d'image

13) *Comprendre l'interpolation numérique, VIRUSPHOTO*
http://www.virusphoto.com/752-comprendre-linterpolation-numerique.html

14)*Class ConvolveOp*
http://docs.oracle.com/javase/7/docs/api/java/awt/image/ConvolveOp.html

15) *Class AffineTransformOp*
http://docs.oracle.com/javase/7/docs/api/java/awt/image/AffineTransformOp.html

16) *Algorithme de Bresenham*
http://rvirtual.free.fr/programmation/OpenGl/Sdin/c1102.htm

D. Outils de codage

16) Java™ Platform, Standard Edition 7 - API Specification
http://docs.oracle.com/javase/7/docs/api/